CptS355 - Assignment 4 (PostScript Interpreter - Part 2) Spring 2023

An Interpreter for a Simple Postscript-like Language (SPS)

Weight: The entire interpreter project (Part 1 and Part 2 together) will count for 12% of your course grade. Part 2 is worth 9%.

This assignment is to be your own work. Refer to the course academic integrity statement in the syllabus.

Download Starter Files

All the starter files provided for this assignment are attached to the HW4 Assignment 4 – Part2 Canvas dropbox as a zip file. The zip file includes several files, but all of your changes will be made to only four files: `psparser.py` , `psexpressions.py`, `psoperators.py`, and `tests_part2.py`.

Here are all the files included in the archive:

1. `repl.py`: implements the REPL for the interpreter
2. `load.py`: implements the loader for the interpreter. Loads and evaluates a collection of PostScript code examples.
3. `psparser.py`: implements the lexer and parser for PostScript input.
4. `psexpressions.py`: defines the classes which represent expressions (`` `Expr` `` and its subclasses) and evaluated values (`` `Value` `` and its subclasses).
5. `psoperators.py`: (not included) You will copy your part1 implementation of `psoperators.py` to the `HW4_part2` folder.
6. `buffer.py`: implements the `Buffer` class, used in `psparser.py`
7. `utils.py`: utility functions used in this assignment
8. `tests_part2.py`: Python unit tests for testing the interpreter output.
9. `colors`.py : Color values you can use when you print in REPL environment.

Turning in your assignment

Copy your `psoperators.py` file from part-1 to the `HW4_part2` folder and continue developing your code. I strongly encourage you to save a copy periodically so you can go back in time if you really mess something up. To submit your assignment, zip all the files as `HW4_part2.zip` and turn in your zip file by uploading on the dropbox on Canvas. Please don't zip the `HW4_part2` directory; your zip archive should only include the above 9 files. Also, exclude any directory that includes binary python bytecode (for example `__pycache__`).

The file that you upload must be named HW4_part2.py . At the top of the `psoperators.py` file in a comment, please include your name and the names of the students with whom you discussed any of the problems in this homework. This is an individual assignment and the final writing in the submitted file should be ***solely yours***. You may NOT copy another student's code or work together on writing code. You may not copy code from the web, or anything else that lets you avoid solving the problems for yourself.

You may turn in your assignment up to 3 times. Only the last one submitted will be graded.

Implement your code for Python 3. The TA will run all assignments using Python3 interpreter. You will lose points if your code is incompatible with Python 3.

The Problem

In this assignment you will write an interpreter in Python for a **simplified** PostScript-like language, concentrating on key computational features of the abstract machine, omitting all PS features related to graphics, and using a somewhat-simplified syntax. The simplified language, SPS, has the following features of PostScript language:

Constants:

- ***integer constants***, e.g. `1, 2, 3, -4, -5.` We will represent integer constants as Python "`int`" values in `opstack` and `dictstack`.
- ***boolean constants***, e.g. , `true , false.` We will represent boolean constants as Python "`bool`" values in `opstack` and `dictstack`.
- ***string constants***, e.g. `(CptS355)` : string delimited in parenthesis. We will represent string constants as `StringValue` objects (see psexpressions.py file.) (Make sure to keep the parenthesis delimiters around the string value when you initialize the `StringValue` object.)
- ***name constants***, e.g. `/fact, /x.` A name constant starts with a '/' and letter followed by an arbitrary sequence of letters and numbers. We will represent name constants as Python "`str`" values in `opstack` and `dictstack`.
- ***names*** to be looked up in the dictionary stack, e.g., `fact, x;` as for name constants, without the '/'
- ***code constants (i.e., code-arrays);*** code between matched curly braces { ... }

Operators:

- ***built-in operators on numbers***: `add, sub, mul, mod, eq, lt, gt`
- ***string creation operator***: string; pops an integer value (e.g., n) from the operand stack and creates a `StringValue` object with `'value'` of length `'n'`. Initializes the characters in the `StringValue`'s `'value'` with '\0' , i.e., ascii NUL character.
- ***built-in operators on string values***: `get, put, getinterval, putinterval, length.` These operators should support `StringValue` arguments.
- ***dictionary creation operator***: `dict`; pops an integer value from the operand stack and creates a new empty dictionary on the operand stack (we will call this `psDict`). `psDict` ignores the popped integer value.
- ***built-in operators on dictionary values***: `get, put, length.` These operators should support `DictionaryValue` arguments.
- ***built-in conditional operators***: `if, ifelse` (you will implement `if/ifelse` operators in Part2)
- ***built-in loop operator***: `for` (you will implement `for` operator in Part 2).
- ***stack operators***: `dup, copy, count, pop, clear, exch, stack`
- ***dictionary stack manipulation operators*** : `begin, end.`
    - `begin` requires one dictionary operand on the operand stack; `end` has no operands.
- ***name definition operator***: `def.` We will call this `psDef.`
- ***stack printing operator***: `stack.` Prints contents of `opstack` without changing it.

Part 2 - Requirements

The starter code provided to you partially implements a REPL (read-evaluate-print-loop) tool for interpreting PostScript code. In Part 2 you will complete the implementation of the following pieces in the starter code:

I. *The Reader*: You need to complete the reader in `psparser.py`. You will convert the PostScript tokens to their corresponding expression objects.
II. *The Evaluator*: You need to implement the `` `eval` `` methods of expression objects in `psexpressions.py`.
III. *Executing (applying) code-arrays:* You need to implement the apply method of `CodeArrayValue` (in `psexpressions.py`). You will also implement **`if`, `ifelse`** operators (`psIf` and `psIfelse`), and **`for`** loop operator (`psFor`). You will write these functions in the `psoperators.py` file from Part 1.

Running the Interpreter

1. **Read-Eval-Print.** You can run the interpreter in the REPL mode and interpret the PostScript code on the command prompt. The RELP tool is implemented in `reply.py` file. The interpreter reads PostScript expressions, evaluates them, and displays the results. You can run the REPL tool by executing the `reply.py` file, i.e.,

```
python repl.py     (or python3 repl.py)
```

2. **Load.** Alternatively, you can run the interpreter on the "Loader" mode and run the loaded PostScript programs all at once. The "Loader" loads PostScript code (given as strings), evaluates them, and displays the results. You can run the "Loader" tool by running the following command on the command line:

```
python load.py     (or python3 load.py)
```

Implementation Overview

Here is a brief overview of each of the Read-Eval-Print Loop components in our interpreter. Refer to this section as you work through the project as a reminder of how all the small pieces fit together!

Read: This step parses user input (a string of PostScript code) into our interpreter's internal Python representation of PostScript expressions.
Lexical analysis has already been implemented in the `tokenize` function in `psparser.py`. This function returns a list of tokens. The `` `read` `` function turns this list into a `` `Buffer` `` (defined in `buffer.py`) which allows to get the tokens one at a time.
Syntactic analysis happens in `` `read` `` and `` `read_expr` `` functions. Together, these mutually recursive functions parse tokens and converts them into our interpreter's internal Python representation of PostScript expressions.
In our interpreter, there are four types of expressions. `psexpressions.py` defines the classes that will represent each of these expressions and they are all subclasses of the `` `Expr` `` class:

1. `Literal`: represents primitive constants : integers or booleans . The `` `value` `` attribute contains the constant value the `` `Literal` `` refers to.
2. `PSName`: represents names of variables, functions, or operators . The `` `var_name` `` attribute contains the name of the variable as a Python string, e.g., `'/sq'`,`'sq'`,`'add'`,`'def'`. If the

`var_name` starts with a `/` character, PSName represents a name constant, otherwise it represents a variable reference , function call, or a built-in operator call.

3. `PSString`: represents strings. The `value` attribute is a Python string enclosed in PostScript string delimiters, i.e., '(' and ')'. For example: `'(CptS355)'`
4. `PSCodeArray`: represents body of a function or blocks of `if`, `ifelse`, and `for` operators. The `value` attribute is a Python list that includes the tokens of the PostScript code-array (block) it represents , e.g., `[Literal(10), Literal(5),PSName(mul)]`

The `read` function calls the `read_expr` function for each token in the buffer and returns a list of expression objects. You will complete `read_expr` function in `psparser.py`. `read_expr` creates and returns the expression object (i.e., one of the above) for the given token.

In `repl.py` and `load.py` , the interpreter calls `read` function to parse the tokens in PostScript code and turns them into a list of expressions.

Eval: This step evaluates PostScript expressions (represented in Python) to obtain values.

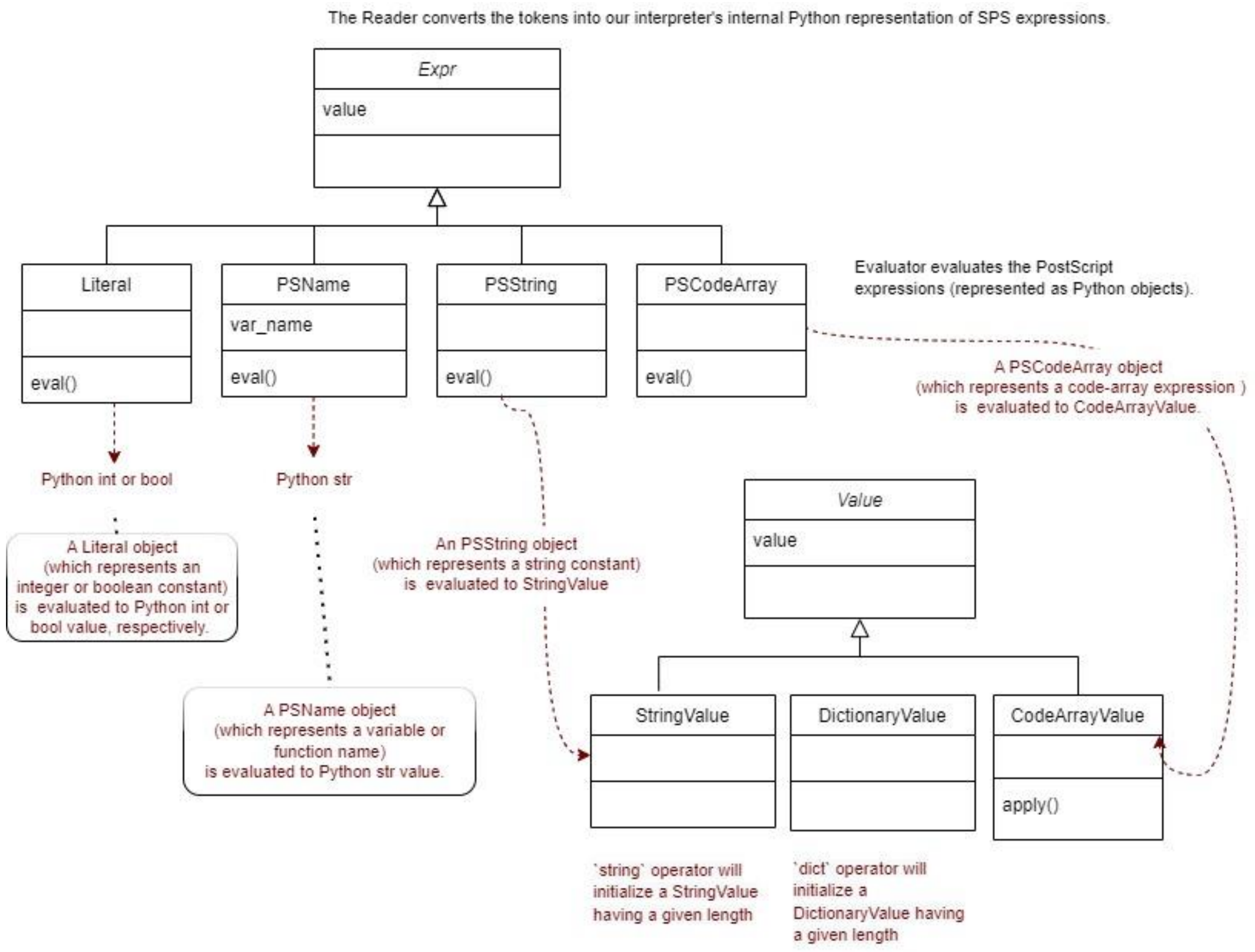
Each of the expression classes (i.e., `Literal`, `PSName`, ` PSString`, and `PSCodeArray`) implement their own `eval` methods returning the value that the expression evaluates to. You will complete the implementations of the `eval` methods for each of these classes in `psexpressions.py.` See “II. The Evaluator” section below for more details.

In `repl.py` and `load.py`, the interpreter calls the `eval` method of each expression to **evaluate** the expressions in the expression list.

Print: This step prints the __str__ representation of the obtained value.

Loop: The logic for the loop is handled by the while loop in `repl.py`. You don’t need to make any changes in `repl.py`.

The UML diagram on the next page illustrates the class level design for the interpreter.


The Reader converts the tokens into our interpreter's internal Python representation of SPS expressions.
Expr
value
Literal
eval()
PSName
var_name
eval()
PSString
eval()
PSCodeArray
eval()
Evaluator evaluates the PostScript expressions (represented as Python objects).
Python int or bool
Python str
A Literal object (which represents an integer or boolean constant) is evaluated to Python int or bool value, respectively.
A PSName object (which represents a variable or function name) is evaluated to Python str value.
An PSString object (which represents a string constant) is evaluated to StringValue
A PSCodeArray object (which represents a code-array expression) is evaluated to CodeArrayValue.
Value
value
StringValue
DictionaryValue
CodeArrayValue
apply()
`string` operator will initialize a StringValue having a given length
`dict` operator will initialize a DictionaryValue having a given length

I. The Reader

The first part of this assignment deals with reading and parsing SPS code. The reader is implemented in `psparser.py`. To parse the SPS programs,

1. We first pass the SPS code to the lexer which convert the continuous input text to a list of tokens. `tokenize` function in `psparser.py` implements the tokenizer. You do not need to make any changes in this part.
   Example: Given the PostScript code ,
```
"""
 /square {dup mul} def
 /mydict 1 dict def
 mydict /in 1 put
 mydict /out 100 put
 mydict /in 10 put
 mydict /in get
 square
 mydict /out get
 eq
 {(equal)} {(different)} ifelse
"""
```

`tokenize` will return :

```
['/square', '{', 'dup', 'mul', '}', 'def', '/mydict', 1, 'dict', 'def', 'mydict',
'/in', 1, 'put', 'mydict', '/out', 100, 'put', 'mydict', '/in', 10, 'put',
'mydict', '/in', 'get', 'square', 'mydict', '/out', 'get', 'eq', '{', '(',
'equal', ')', '}', '{', '(', 'different', ')', '}', 'ifelse']
```

2. Next, we parse the tokens, i.e., convert the tokens into our interpreter's internal Python representation of SPS expressions. The `read` and `read_expr` functions in psparser.py handle parsing.
   - The `read` function turns the token list (the output of `tokenize`) into a `Buffer` object (defined in buffer.py) which allows to get the tokens one at a time.

```
"""Parse an expression from a string. If the string does not contain an
   expression, None is returned. If the string cannot be parsed, a SyntaxError
   is raised.
"""
def read(s):
    #reading one token at a time
    src = Buffer(tokenize(s))
    out = []
    while src.current() is not None:
        out.append(read_expr(src))
    return out
```

- `read` calls the `read_expr` function to convert each expression to its corresponding expression object. You need to complete the `read_expr` function and create the corresponding expression object for each token.
  - As explained in section "Implementation Overview", in our interpreter there are four types of expressions: Literal, PSName, PSString, and PSCodeArray which are subclasses of the `Expr` object defined in psexpressions.py.
  - In the `read_expr` function, you can use the helper functions provided in psparser.py to check the type of each token and to retrieve the tokens that will be included in the array or code-array values. Note that the tokens between `(` and `)` belong to a string and those between `{` and `} belong to a code array.

```
""" Converts the next token in the given Buffer to an expression. """
def read_expr(src):
    token = src.pop_first()
    if token is None:
        raise SyntaxError('Incomplete expression')
    # TO-DO  - complete the following; include each condition as an `elif` case.
    #   if the token is a literal return a `Literal` object having `value` token.
    #   if the token is a name, create a PSName object having `var_name` token.
    #   if the token is an array delimiter (i.e., '('), get all tokens until the
    #     matching ')' delimiter and combine them as a Python string
    #     separated by space; create a StringValue object having this string value.
    #   if the token is a codearray delimiter (i.e., '{'), get all tokens until
    #       the matching '}' delimiter and combine them as a Python list;
    #       create a PSCodeArray object having this list value.
    else:
        raise SyntaxError("'{}' is not the start of an expression".format(token))
```
Given the tokens,

```
['/square', '{', 'dup', 'mul', '}', 'def', '/mydict', 1, 'dict', 'def', 'mydict',
'/in', 1, 'put', 'mydict', '/out', 100, 'put', 'mydict', '/in', 10, 'put',
'mydict', '/in', 'get', 'square', 'mydict', '/out', 'get', 'eq', '{', '(',
'equal', ')', '}', '{', '(', 'different', ')', '}', 'ifelse']
```

\`read\` should return:

```
[PSName(/square), PSCodeArray([PSName(dup), PSName(mul)]), PSName(def),
PSName(/mydict), Literal(1), PSName(dict), PSName(def), PSName(mydict), PSName(/in),
Literal(1), PSName(put), PSName(mydict), PSName(/out), Literal(100), PSName(put),
PSName(mydict), PSName(/in), Literal(10), PSName(put), PSName(mydict), PSName(/in),
PSName(get), PSName(square), PSName(mydict), PSName(/out), PSName(get), PSName(eq),
PSCodeArray([PSString((equal))]), PSCodeArray([PSString((different))]),
PSName(ifelse)]
```

The above is the print of the __repr__ representation of the expression list \`read\` returns.

II. The Evaluator

Evaluator evaluates the PostScript expressions (represented as Python objects). In `repl.py` and `load.py`, the interpreter calls the \`eval\` method of each expression to **evaluate** the expressions in the expression list.
Each of the expression classes (i.e., `Literal`, PSName, PS`String`, and PSCode`Array`) implement their own \`eval\` method returning the value that the expression evaluates to. For example:

1. \`Literal\` object is evaluated to the integer or boolean value it represents (e.g. 15 or \`True\` or \`False\`) and pushed onto the `opstack`.
2. \`PSName\` object is evaluated according to the following:
   a. If the \`PSName\` represents a name constant (i.e., its \`var_name\` attribute starts with a \`/\`), it will be evaluated to a Python string having value \`var_name\`. The evaluated value will be pushed onto the `opstack`.
   b. If the \`PSName\` represents a built-in operator (i.e., its \`var_name\` attribute is one of the built-in operator names), then we will evaluate it by executing the operator function defined in `psoperators.py` in the current environment (`opstack`).
   c. If the \`PSName\` represents a variable or function, interpreter looks up the value of the variable in the current environment (`dictstack`).
      i. If the variable value is a function (\`CodeArrayValue\`), it should be applied (i.e., executed) by calling its \`apply\` method.
      ii. Otherwise, the variable value is a constant and it should be pushed onto the `opstack`.
3. \`PSString\` object is evaluated to a \`StringValue\` value. For example: a PS`String` with \`value\` attribute '(CptS355)' will be evaluated to `StringValue` with \`value\` attribute '(CptS355)'. The evaluated `StringValue` is pushed onto the stack.
4. \`PSCodeArray\` object is evaluated to \`CodeArrayValue\` value. For example: a \`PSCodeArray\` with \`value\` attribute `[PSName(dup), PSName(mul)]` will be evaluated to \`CodeArrayValue\` with the same \`value\` (i.e., `[PSName(dup), PSName(mul)]`. The evaluated \`CodeArrayValue\` is pushed onto the stack.

III. Handling of code-arrays ; `if/ifelse`, `for` operators

Recall that a `PSCodeArray` represents the blocks of `if`, `ifelse`, and `for` operators and the function bodies. As explained above, when a `PSCodeArray` is evaluated, a `CodeArrayValue` object having the same list of tokens is pushed to the `opstack`. Once a `CodeArrayValue` is on the stack several things can happen:

- if it is the top item on the stack when a `def` is executed (i.e. the `CodeArrayValue` is the body of a function), it is stored as the value of the name defined by the `def`.
- if it is the body part of an `if/ifelse` operator, the `psIf` (or `psIfElse`) function will execute (apply) the `CodeArrayValue`. For the `if` operator, the `CodeArrayValue` is executed only if the "condition" argument for `if` operator is true. For the `ifelse` operator, if the "condition" argument is true, first `CodeArrayValue` is executed, otherwise the second `CodeArrayValue` is executed .
- if it is the body part of a `for` operator, `psFor` function will execute (apply) the `CodeArrayValue` as part of the evaluation of the `repeat` loop.
- finally, when a `PSName` is looked up and if its value is a `CodeArrayValue`, then the expression represents a function call and the `CodeArrayValue` value will be executed (applied).

A `CodeArrayValue` can be executed by calling its `apply` method. `apply` should evaluate all tokens in the function body.

Testing

1. Testing your interpreter manually:

You can do manual tests by using the RELP tool, i.e., by typing PostScript code on the REPL command line and checking the contents of the `opstack` and `dictstack` after the PostScript code is interpreted. I suggest you print both the `opstack` and the `dictstack` when the "`stack`" operation called to help with the debugging. You can print the `opstack` and `dictstack` in different colors to improve the readability. `colors.py` file includes some predefined colors. To print a string `s` in green and then reset the color back to default, you can use the print statement : `print(OKGREEN+s+CEND)`

2. Testing your interpreter using the loader:

You can load the PostScript code in the loader and interpret it. The given test inputs are already included in `load.py`. The `for` loop in this file iterates over each test input included in `tests` list, interprets them, and print the content of the `opstack` . When you add a new PostScript test input to this file, make sure to add it to the `tests` list. The expected outputs for the given 32 tests are included in the "`expected_output`" dictionary in `load.py`.

3. Automated tests:

Sample Python unittests testing for the same inputs are included in `tests_part2.py` file. **You should provide 5 additional Python unittests in addition to the provided tests.** Make sure that the PostScript code in your tests are different than the provide examples and your tests include several operators. You will loose points if you fail to provide tests or if your tests are too simple.

Testing the parsing:

Before even attempting to run your full interpreter, make sure that your parsing is working correctly. Make sure you get the correct parsed output for the testcases. The parsed tokens for the 31 tests are provided in load.py – see the "`parse_output`" dictionary in `load.py`.